10 Common OAuth token vulnerabilities

This chapter covers

- What a bearer token is and how to safely generate it
- Managing the risk of using bearer tokens
- Safely protecting bearer tokens
- What an authorization code is and how to safely handle it

In the previous chapters, we've analyzed implementation vulnerabilities that affected all the actors of an OAuth deployment: clients, protected resources, and authorization servers. Most of the attacks we've seen had a single purpose: to steal an access token (or an authorization code used to get an access token). In this chapter, we go deeper into what it takes to create good access tokens and authorization codes, and what we can do to minimize the risks while handling them. We're going to look at what happens when the token is stolen, and we'll see how this can cause relatively minor damage compared with a hijacked password. In summary, the motivations behind OAuth are to offer a more safe and flexible model compared with the password-driven world.

10.1 What is a bearer token?

One choice made by the OAuth working group while designing the OAuth 2.0 specification was to drop the custom signature mechanism present in the original OAuth 1.0 specification in favor of relying on secure transport-layer mechanisms,

such as TLS, between parties. By removing the signing requirement from the base protocol, OAuth 2.0 can accommodate different kinds of tokens. The OAuth specification defines a *bearer token* as a security device with the property that any party in possession of the token (a "bearer") can use the token, regardless of who that party is. In this way, a bearer token is much like a bus token or a ticket to an amusement park ride: these items grant access to services, and don't care who uses them. As long as you've got a bus token, you can ride the bus.

From a technological standpoint, you can think about bearer tokens in much the same way as you do browser cookies. Both share some basic properties:

- They use plaintext strings.
- No secret or signature is involved.
- TLS is the basis of the security model.

But there are some differences:

- Browsers have a long history of dealing with cookies, whereas OAuth clients don't.
- Browsers enforce the same origin policy, meaning that a cookie for one domain isn't passed to another domain. This isn't the case for OAuth clients (and may be a source of problems).

The original OAuth 1.0 protocol called for tokens that also had an associated secret, which was used to calculate a signature across the request. This signature was then verified by the protected resource alongside the token value itself, proving possession of both the token and its associated secret. Calculating this signature correctly and consistently turned out to be a big burden for client and server developers, and the process was prone to many frustrating errors. Calculating the signature depended on many factors, such as encoding string values, ordering the request parameters, and canonization of the URI. In combination with the fact that cryptography doesn't forgive even the smallest mistake, things constantly broke because of mismatching signatures.

For instance, a server-side application framework could inject or reorder parameters, or a reverse proxy could hide the original request URI from the application doing the OAuth processing. One of the authors knows first hand of a developer whose OAuth 1.0 implementation used an uppercase hex encoding on the client side (as %3F, %2D, and %3A), and lowercase hex encoding on the server side (as %3f, %2d, and %3a). This particular implementation bug was infuriating to discover. Although a human can easily see these as equivalent, and any machine interpreting the hex value can easily transform it, cryptographic functions require an exact match between both sides in order for the signature to validate properly.

Furthermore, the requirement for TLS never went away. Without TLS for the get-a-token step, the access token and its secret could be stolen. Without TLS for the use-a-token step, the results of the authorized call could be stolen (and sometimes replayed within a time window). As a result, OAuth 1.0 had the reputation of being a complicated, difficult-to-use protocol. The new OAuth 2.0 specification was formed

with bearer tokens at the center of a new simplified protocol. Message-level signatures weren't entirely abandoned, but merely set aside. With time, some users of the OAuth 2.0 specification asked for an extension of the protocol that included some sort of signature; we'll meet some alternatives to bearer tokens in chapter 15.

10.2 Risks and considerations of using bearer tokens

Bearer tokens have characteristics similar to the session cookies used in browsers. Unfortunately, misunderstanding this parallelism leads to all sorts of security problems. When an attacker is able to intercept an access token, they are able to access all resources covered by the scope of that particular token. A client using a bearer token doesn't need to prove possession of any additional security items, such as cryptographic key material. Apart from token hijacking (which we've covered in depth in many parts of this book), the following threats associated with OAuth's bearer tokens are common to many other token-based protocols:

Token forgery. An attacker may manufacture its own bogus token or modify an existing valid one, causing the resource server to grant inappropriate access to the client. For example, an attacker can craft a token to gain access to information they weren't able to view before. Alternatively, an attacker could modify the token and extend the validity of the token itself.

Token replay. An attacker attempts to use an old token that was already used in the past and is supposed to be expired. The resource server shouldn't return any valid data in this case; instead, it should return an error. In a concrete scenario, an attacker legitimately obtains an access token in the first place and they'll try to reuse it long after the token has expired.

Token redirect. An attacker uses a token generated for consumption by one resource server to gain access to a different resource server that mistakenly believes the token to be valid for it. In this case, an attacker legitimately obtains an access token for a specific resource server and they try to present this access token to a different one.

Token disclosure. A token might contain sensitive information about the system and the attacker is then something that they couldn't know otherwise. Information disclosure can be considered a minor problem compared with the previous one, but it's still something we need to care about.

The foregoing are all severe threats that apply to tokens. How can we protect bearer tokens at rest and in transit? Security as an afterthought never works, and it's important the implementer makes the right choices in the early phase of any project.

10.3 How to protect bearer tokens

It is of extreme importance that access tokens sent as bearer tokens are not sent in the clear over an insecure channel. As per the core OAuth specification, transmission of access tokens must be protected using end-to-end confidentiality, such as SSL/TLS.

What is SSL/TLS then? Transport Layer Security (TLS), formerly known as Secure Sockets Layer (SSL), is a cryptographic protocol designed to provide communications security over a computer network. The protocol protects transmissions between two parties directly connected to each other, and the encryption process has the following aspects:

- The connection is private because symmetric cryptography is used to encrypt the data transmitted.
- The connection is reliable because each message transmitted includes a message integrity check using a message authentication code.

This is achieved typically by using certificates with public key cryptography; in particular, on the public internet, the application initiating the connection request verifies the certificate of the application receiving the connection request. In some limited circumstances, the certificate of the application initiating the connection request can also be verified, but such *mutual authentication* of the TLS connection is fairly limited and rare. It is important to remember that OAuth bearer tokens can't be used securely without TLS being part of the connection in order to protect them in transit.

Where's all the TLS?

You've probably noticed by now that in all of our lab exercises, we haven't used TLS at all. Why would we do that? The deployment of a fully secure TLS infrastructure is a complex topic, far beyond the scope of this book, and getting TLS working isn't required for understanding how the core mechanics of OAuth work. As with authenticating the resource owner, which is also required for a functional and secure OAuth system, in our exercises we're leaving this out for the sake of simplicity. In a production system, or any deployment in which you care about the security of the components, proper TLS usage is a hard-and-fast requirement.

Remember, implementing secure software is something that needs to be done right every time, whereas the hackers only have to get things right once.

In the following sections, we'll see what the different OAuth components can do to deal with threats associated with bearer tokens.

10.3.1 At the client

We've seen in various parts of this book how access tokens can be stolen from client applications and revealed to the attacker. We need to remember that bearer access tokens are transparent for clients and there isn't any cryptographic operation they need to perform. Hence, when an attacker obtains a bearer access token, they're able to access all the resources associated with the token and its scope.

One countermeasure that a client can apply is to limit the scope of the token to the minimum required for its tasks. For example, if all the client needs to achieve its purpose is the resource owner's profile information, it would be enough to ask for the

`profile` scope (and not any other scope, for example, `photo` or `location`).[1] This approach of "minimal privilege" limits what the token can be used for if it's captured. To minimize impact on the user experience, a client can ask for all appropriate scopes during the authorization phase, then use the refresh token to get limited-scope access tokens to call the resource directly.

It would also be beneficial, if feasible, to keep access tokens in transient memory to minimize attacks derived from repository injections. Doing so even if the attacker is able to get their hands on the client's database won't gain any information regarding access tokens. This isn't always feasible for all client types, but secure storage of tokens, away from the prying eyes of other applications and even end users, is something that every OAuth client application should be doing.

10.3.2 At the authorization server

If an attacker is able to gain access to the authorization server database or launch a SQL injection against it, then the security of multiple resource owners might be compromised. This happens because the authorization server is the central point that coordinates and emits access tokens, issued to multiple clients and potentially consumed by multiple protected resources. In most implementations, including our own so far, the authorization server stores access tokens in a database. The protected resource validates them upon receipt from a client. This can be achieved in multiple ways but typically a query is launched against the data looking for the matching token. In chapter 11, we'll see an alternative stateless approach based on structured tokens: JSON Web Tokens, or JWT.

As one efficient precaution, the authorization server can store hashes of the access token (for example, using SHA-256) instead of the text of the token itself. In this case, even if the attacker was able to steal the entire database containing all the access tokens, there isn't much it can do with the information leaked. Although hash salting is recommended for storage of user passwords, it should not be required to use additional salt because the access token value should already include a reasonable level of entropy in order to make offline dictionary attacks difficult. For instance, with a random-value token, the token value should be at least 128 bits long and constructed using a cryptographically strong random or pseudorandom number sequence.

In addition, it would be good to keep access token lifetimes short in order to minimize the risk associated with the leak of a single access token. This way, even if a token is compromised, its valid lifetime limits its usefulness to the attacker. If a client needs to have longer access to a resource, the authorization server can issue a refresh token to the client. Refresh tokens are passed between the client and authorization server, but never the protected resource, limiting the attack surface for this style of long-lived token significantly. The definition of what constitutes a "short' token lifetime depends entirely on the application being protected, but generally speaking, the token shouldn't live much longer than it will be needed for average use of an API.

[1] https://bounty.github.com/researchers/stefansundin.html

Ultimately, one of the best things that can be done at the authorization server is pervasive and secure auditing and logging. Whenever a token is issued, consumed, or revoked, the context in which that took place (the client, resource owner, scopes, resource, time, and so on) can be used to watch for suspicious behavior. As a corollary, all of these logs must be kept clear of the access token values to keep them from leaking.

10.3.3 At the protected resource

The protected resource often handles access tokens in a way similar to that of the authorization server, and should be treated with the same care for security. Since there are likely to be more protected resources than authorization servers on a network, perhaps even more direct care should be given. After all, if you're using bearer tokens, there is nothing stopping a malicious protected resource from replaying an access token to other protected resources. Keep in mind that access tokens can inadvertently leak in system logs, especially those that capture all incoming HTTP traffic for analysis. Tokens should be scrubbed from such logs so that the token value isn't used there.

A resource endpoint should be designed to limit token scope, respecting the collection minimization principle and asking only for the minimum set of scopes needed to handle a particular job. Although it's the clients that request the scopes associated with a token, designers of protected resources can protect the ecosystem by requiring tokens with only the most specific set of scopes possible for functionality. This part of the design process partitions the application's resources in logical ways such that a client won't need to ask for more functionality than necessary in order to do its job.

The resource server should also properly validate the token and avoid the use of special-purpose access tokens that have some sort of super power.[2] Although it's common for a protected resource to cache the current status of a token, especially when using a protocol such as token introspection as discussed in chapter 11, the protected resource must always weigh the benefits and drawbacks of such a cache. It's also a good idea to use rate limiting and other techniques to protect the API, which help prevent attackers from fishing for valid tokens at the protected resource.

Keeping access tokens in transient memory is something that will help out in case of attacks against the resource server's data store. This will make it more difficult for an attacker to discover valid access tokens by attacking a back-end system. Granted, in these cases, the attacker will likely have access to the data being protected by the resource, so as always the costs and benefits need to be balanced.

10.4 Authorization code

We've already encountered authorization codes in chapter 2, and we've seen that the biggest benefit of this grant type is the transmission of an access token directly to the client without passing it through the resource owner's user-agent and potentially exposing it to others, including the resource owner. That said, we've also seen in chapter 7 how some sophisticated attacks can lead to the authorization code being hijacked.

[2] http://www.7xter.com/2015/03/how-i-exposed-your-private-photos.html

The authorization code isn't useful on its own, especially if the client has its own client secret with which it can authenticate itself. However, native applications, as we've seen in chapter 6, have specific problems with client secrets. Dynamic registration, discussed in chapter 12, is one approach to this problem, but it's not always available or appropriate for a given client application. In order to mitigate such attacks against public clients, the OAuth working group released an additional specification that hinders such attack vectors, Proof Key for Code Exchange (PKCE, pronounced "pixie").

10.4.1 Proof Key for Code Exchange (PKCE)

OAuth 2.0 public clients using the authorization code grant are susceptible to the authorization code interception attack. The PKCE specification[3] has been introduced as a way to defend from this attack by establishing a secure binding between the authorization request and the subsequent token request. The way PKCE works is simple:

- The client creates and records a secret named the `code_verifier`, shown in figure 10.1 as a flag with a magic wand on it.
- The client then computes `code_challenge` based on the `code_verifier`, shown in figure 10.1 as the same flag with a complex design overlaid on top of the secret. This can be either the `code_verifier` taken verbatim or the SHA-256 hash of the `code_verifier`, though the cryptographic hash is strongly preferred as it prevents the verifier itself from being intercepted.
- The client sends the `code_challenge` and an optional `code_challenge_method` (a keyword for plain or SHA-256 hash) along with the regular authorization request parameters to the authorization server (see figure 10.1).

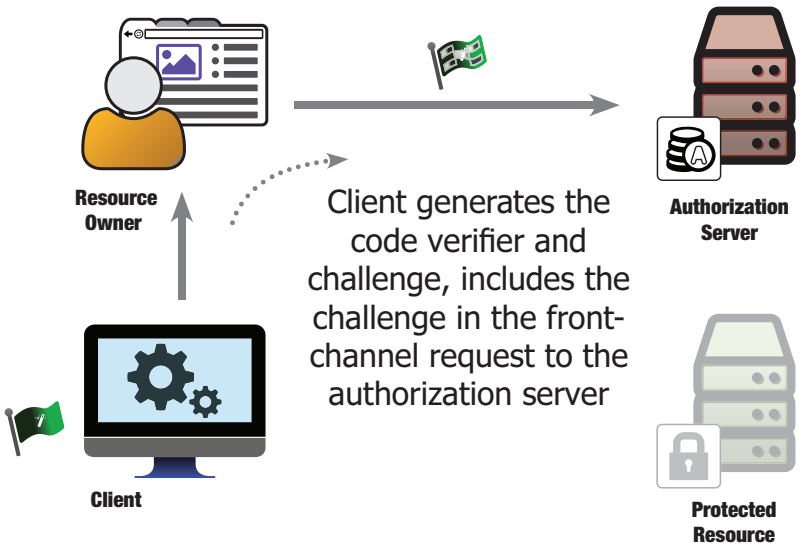

Figure 10.1 PKCE `code_challenge`

[3] RFC 7636 https://tools.ietf.org/html/rfc7636

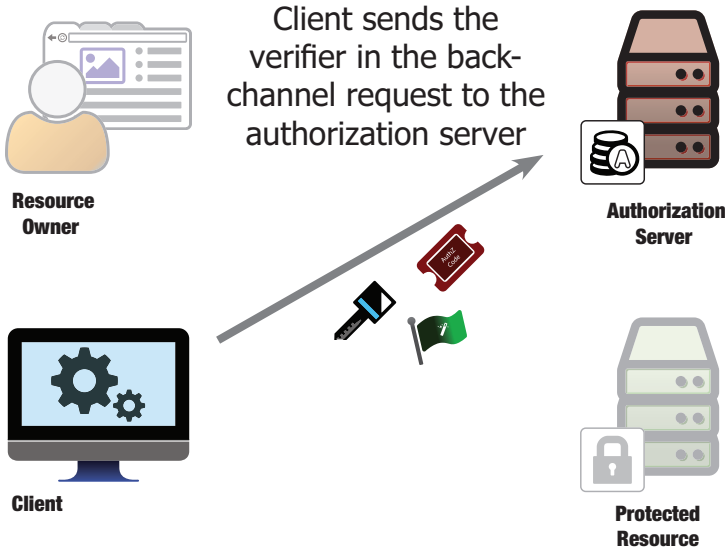

Figure 10.2 PKCE `code_verifier`

- The authorization server responds as usual but records `code_challenge` and the `code_challenge_method` (if present). These are associated with the authorization code that was issued by the authorization server.
- When the client receives the authorization code, it makes a token request as usual and includes the `code_verifier` secret that it previously generated (see figure 10.2).
- The server recomputes the `code_challenge`, and checks to see whether it matches the original (see figure 10.3). An error response is returned if they aren't equal, and the transaction continues as normal if they are.

Adding PKCE support to clients and authorization servers is extremely simple. One of the strengths of PKCE (apart from the undoubted security benefit) is that it can be added on a second stage, without having any interruption of the service, even if the client or the authorization server is in a production environment. We will prove this statement by adding PKCE support to our existing client and authorization server, implementing the `S256` (that uses SHA256) `code_challenge_method`. The `S256` method is mandatory to implement on the server, whereas clients are permitted to use `plain` only if they cannot support `S256` for some technical reason.

Open up the `ch-10-ex-1` folder and edit the `client.js` file. Locate the authorization request section. There we need to generate the `code_verifier`, calculate the `code_challenge`, and send the challenge to the authorization server. The PKCE specification suggests a minimum length of 43 characters and a maximum length of 128 characters for the `code_verifier`. We've chosen to generate a conservative string with length 80. We're using the `S256` method to hash the code verifier.

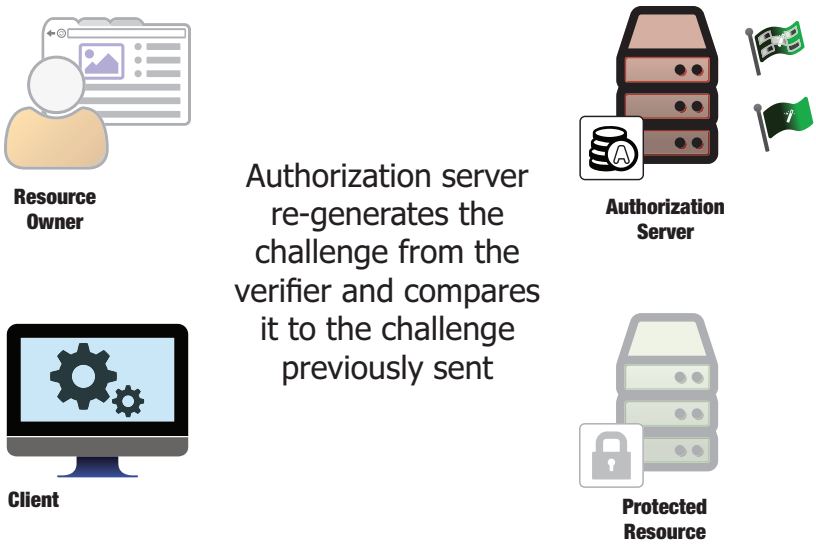

Figure 10.3 Comparing the `code_verifier` with the `code_challenge`

```
code_verifier = randomstring.generate(80);
var code_challenge = base64url.fromBase64(crypto.createHash('sha256').
   update(code_verifier).digest('base64'));

var authorizeUrl = buildUrl(authServer.authorizationEndpoint, {
   response_type: 'code',
   scope: client.scope,
   client_id: client.client_id,
   redirect_uri: client.redirect_uris[0],
   state: state,
   code_challenge: code_challenge,
   code_challenge_method: 'S256'
});
res.redirect(authorizeUrl);
```

Now we also need to modify the `/callback` endpoint in order to pass the `code_verifier` along with the authorization `code` in our call to the token endpoint.

```
var form_data = qs.stringify({
   grant_type: 'authorization_code',
   code: code,
   redirect_uri: client.redirect_uri,
   code_verifier: code_verifier
});
```

Once we are done with the client, we can tackle the server. Because our authorization server stores the original request to the authorization endpoint along with the authorization code, we don't need to do anything special to save the code challenge for later. We'll be able to pull it out of the `code.request` object when necessary. However, we do need to verify the request. In the `/token` endpoint, compute a new `code_challenge` based on the `code_verifier` that was sent to us and the `code_challenge_method`

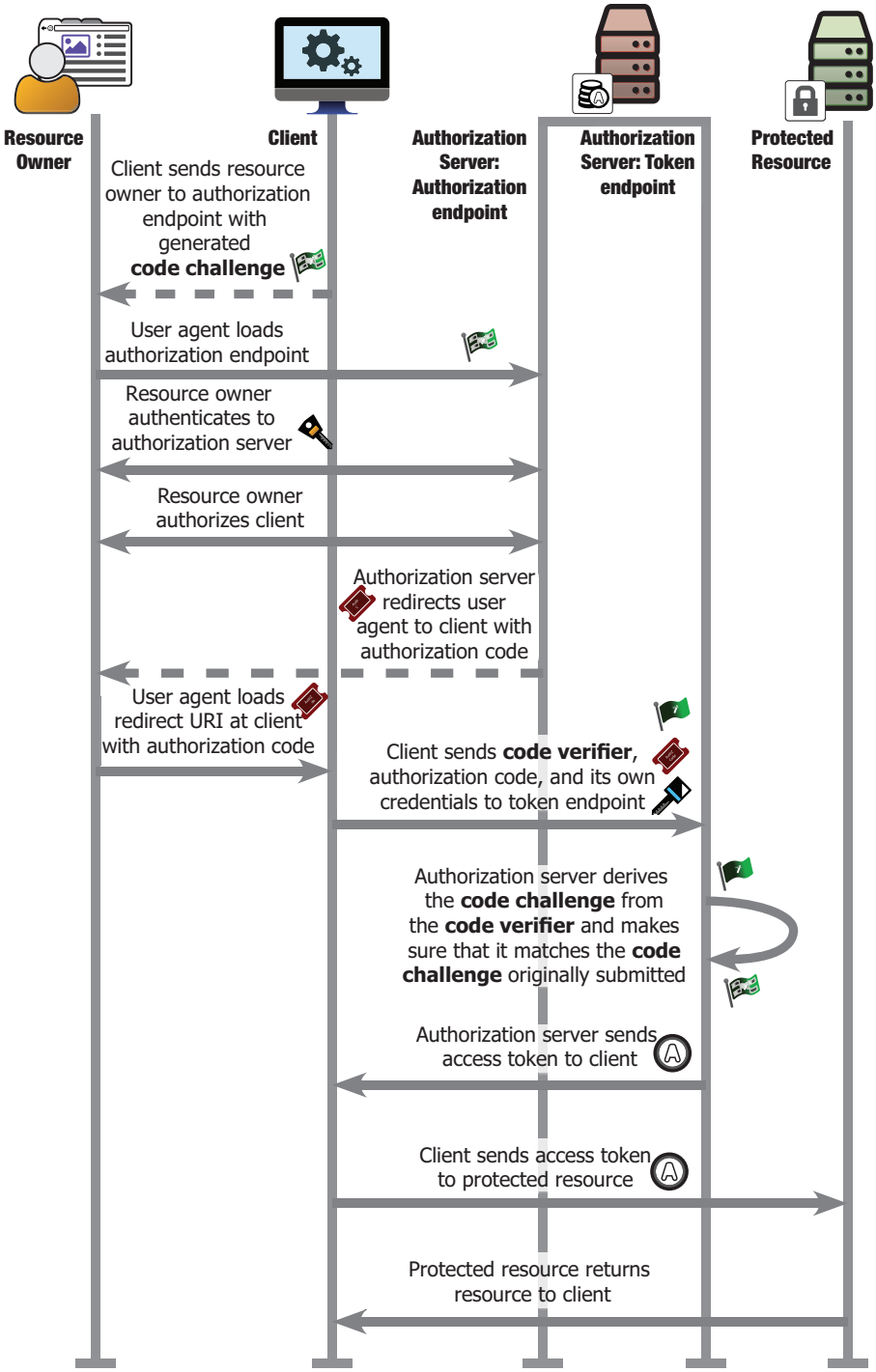

Figure 10.4 PKCE detailed view

that was sent as part of the original request. Our server is going to support both the `plain` and `S256` methods. Notice that the `S256` method uses the same transformation that the client used to generate the `code_challenge` in the first place. We can then make sure that the recomputed `code_challenge` matches the original, and return an error if this doesn't happen.

```
if (code.request.client_id == clientId) {
   if (code.request.code_challenge) {

        if (code.request.code_challenge_method == 'plain') {
               var code_challenge = req.body.code_verifier;
        } else if (code.request.code_challenge_method == 'S256') {
               var code_challenge = base64url.fromBase64(crypto.
   createHash('sha256').update(req.body.code_verifier).digest('base64'));
        } else {
               res.status(400).json({error: 'invalid_request'});
               return;
        }

        if (code.request.code_challenge != code_challenge) {
               res.status(400).json({error: 'invalid_request'});
               return;
        }
   }
```

If everything matches up, we return a token as normal. Notice that even though PKCE is intended for use with public clients, confidential clients can use this method as well. Figure 10.4 shows a full, detailed view of the PKCE flow.

10.5 Summary

Bearer tokens provide a powerful simplification of the OAuth process, allowing developers to more easily and correctly implement the protocol. But with that simplicity comes requirements to protect the tokens throughout the system.

- Transmission of access tokens must be protected using secure transport layer mechanisms such as TLS.
- The client should ask for the minimum information needed (be conservative with the scope set).
- The authorization server should store hashes of the access token instead of clear text.
- The authorization server should keep access token lifetime short in order to minimize the risk associated with the leak of a single access token.
- The resource server should keep access tokens in transient memory.
- PKCE may be used to increase the safety of authorization codes.

At this point, we've built an entire OAuth ecosystem from the ground up and dug into the vulnerabilities that can come from bad implementations and deployments. Now we're going to take a bit of a step outside of OAuth itself and look at the wider ecosystem of capabilities.